RGB Alpha Saturation Using Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243654-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) provide floating point Single-Instruction, Multiple-Data (SIMD) instructions as well as additional SIMD integer instructions and Cache Control Instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note describes a color saturation algorithm, RGB alpha saturation, and presents key optimizations to improve the overall performance using the Streaming SIMD Extensions.

Although most of the Streaming SIMD Extensions are focused on enhancing floating point calculations, this application note highlights the benefits of the additional SIMD integer instructions and the Cache Control Instructions. The RGB alpha saturation algorithm is an integer based algorithm. Even applications that have taken full advantage of the original MMX™ technology integer instructions and optimization techniques can gain significant performance improvement with small modifications to the code using the Streaming SIMD Extensions additional SIMD integer instructions and the memory pre-fetch instructions. Examples of code that exploit Streaming SIMD Extensions are included.

2 RGB Alpha Saturation

Red, green, and blue (RGB) are the three primary colors used in video processing, and RGB often refers to the three unencoded outputs of a color camera. The encoded RGB values at a pixel, which is the smallest distinguishable and resolvable area in a video image, is the resulting perceived color. The number of different colors depends on the number of bits used to represent each R, G, and B value. For example, 8-bit R, G, and B values can range between 0 and 255, representing 256 different colors. RGB values of [255,255,255] are perceived as white, and values of [0,0,0] are perceived as black.

Color saturation is the degree to which a color is free of white light. In other words, color saturation controls the intensity or purity of a color. Adding black or white to a color decreases its saturation. Color saturation is used in computer graphics and digital photography applications. Colors that appear to be washed-out or faded are under saturated. If similar colors blend together or appear dark, they are over saturated. In photography, this is equivalent to being overexposed or underexposed.

RGB alpha saturation is the process of limiting each of the three unencoded RGB output streams to a maximum or minimum alpha value. This application note focuses on limiting the RGB values to a maximum alpha value. Applying this same algorithm to the case where the RGB values are limited to the minimum alpha value is straightforward.

2.1 Applications for RGB Alpha Saturation

Desktop publishing, digital video, and computer photography applications are prime examples of where color saturation techniques are utilized to improve image quality. Color images that appear to be washed out or faded can be digitally enhanced by reducing the white light and increasing the purity of the colors. Likewise, brightening a dark image can enhance the color appearance by increasing the white light. Note that RGB alpha saturation is only one technique that can be used to modify the color saturation. Other color models, such as the HLS model (hue, luminance, and saturation) can adjust the color saturation using different techniques.

2.2 Implementing the RGB Alpha Saturation

Alpha saturation is the process of comparing each R, G, B value with an associated alpha value. Any R, G, and/or B value that is greater than alpha is replaced with the alpha value. Note that the algorithm as presented assumes RGBA order when fetched from memory. With minor modification, the algorithm works even if the order is reversed.

The algorithm implemented here is:

A) Prefetch 96 bytes ahead. The reason for prefetching 96 bytes instead of some other value will be discussed in the later section on Software Controlled Data Prefetching.

B) Load two sets of RGBA values into one MMX technology register.

C) Build a mask of the alpha values for each RGBA value in a second MMX technology register.

D) Take the minimum value between the two RGBA values and the alpha masks.

E) Store the result.

F) Repeat Steps A - E for all RGBA values.

A0	B0	G0	R0	A1	B1	G1	R1	where R1 > A1 and G0 > A0
A0	A0	A0	A0	A1	A1	A1	A1	build mask of alphas
A0	B0	A0	R0	A1	B1	G1	A1	result is the minimum of each byte pair

The assembly implementation is given below.

A) Prefetch RGBA values ahead of when they will actually be needed. It will take approximately 4 iterations through the loop before a prefetched value is actually loaded into cache. So RGBA values are prefetched 96 bytes ahead of the current RGBA values being processed. . Again, the reason for prefetching 96 bytes instead of some other value will be discussed in the later section on Software Controlled Data Prefetching

```
prefetchnta mmword ptr[eax + 96]
```

B) Load two sets of RGBA values into one MMX technology register.

```
movq     mm0,  mmword ptr [eax]
```

mm0 —	A0	B0	G0	R0	A1	B1	G1	R1

C) Build a mask of the alpha values for each RGBA value in a second MMX technology register.

```
pshufw   mm1, mm0, 0xF5
```

Note that 245 = 11 | 11 | 01 | 01

mm1 —	A0	B0	A0	B0	A1	B1	A1	B1

After the shuffle word operation, the MMX technology register still contains the B component where alpha values are needed. Complete building the alpha mask by ANDing the MMX technology register with a mask to remove the B components.

```
pand     mm1, 0xff00ff00ff00ff0
```

mm1	A0	B0	A0	B0	A1	B1	A1	B1
mask	ff	00	ff	00	ff	00	ff	00
mm1	A0	00	A0	00	A1	00	A1	00

movq mm2, mm1

mm2	A0	00	A0	00	A1	00	A1	00	Copy the mm1 MMX technology register.

psrld mm1, 8

mm1	00	A0	00	A0	00	A1	00	A1	“shift” MMX technology register.

por mm1, mm2

mm2	A0	00	A0	00	A1	00	A1	00	
mm1	00	A0	00	A0	00	A1	00	A1	“or” the two resulting MMX technology registers.
mm1	A0	A0	A0	A0	A1	A1	A1	A1	The alpha mask is now complete.

D) Take the minimum values between the two RGBA values and the alpha masks.

pminub mm0, mm1

mm2	A0	B0	G0	R0	A1	B1	G1	R1	
mm1	A0	A0	A0	A0	A1	A1	A1	A1	
mm0	A0	B0	A0	R0	A1	B1	G1	A1	where R1 > A1 and G0 > A0.

E) Store the result.

```
movq      mmword ptr [eax-8], mm0
```

This is a small loop with no dependencies between successive iterations, so it is possible to unroll the loop such that four RGBA values are processed within one loop iteration. This can help the pipeline improve the overall out-of-order execution scheduling, and reduce the loop overhead by half the number of iterations.

Note that this algorithm evaluates two RGBA values per iteration and four RGBA values when unrolled. But, the number of RGBA values may not be even or a multiple of four when unrolled. Any implementation of this algorithm must be able to detect and handle these conditions. The implementation using Streaming SIMD Extensions provided in Section 7 includes the code required to handle these conditions.

2.3 MMX™ Technology versus Streaming SIMD Extensions Implementation

Below are two code fragments from the main loops of an implementation of the algorithm using Streaming SIMD Extensions and one using MMX technology. Upon closer inspection of the code

fragments, there are two important differences between the two implementations that greatly affect overall performance. The first is prefetching, which is only available in the Streaming SIMD Extensions instruction set. Prefetching adds one instruction per loop iteration, but can hide a portion of the load latencies for memory accesses by allowing other computational instructions to continue execution before the actual memory load is required. The second is that an implementation using Streaming SIMD Extensions reduces by seven the number of required instructions per loop iteration compared to an implementation using MMX™ technology. The details are as follows:

- Prefetch adds one instruction.
- The shuffle word instruction, to build the mask of alpha values, reduces the number of instructions within the loop by two instructions.
- The minimum unsigned byte instruction set reduces the number of instructions within the loop by another six instructions.

Both the shuffle word and the minimum unsigned byte instructions are only available in the Streaming SIMD Extensions instruction set. Also, unrolling the larger implementation that uses MMX technology does not provide the same benefit achieved with the smaller implementation using Streaming SIMD Extensions. When unrolled, the MMX technology implementation becomes register bound (within the MMX register set). The larger loop iterations require additional instructions and is no longer independent. The additional overhead required is larger than the gain achieved by reducing the loop iteration overhead.

Assuming that the individual instruction execution latencies are approximately the same in both implementations, the version that uses Streaming SIMD Extensions has half the number of instructions and should, therefore, be twice as fast as the version that uses MMX technology.

Implementation using Streaming SIMD Extensions

prefetchnta	[eax + 96]
pshufw	mm1, mm0
pand	mm1, mm7
movq	mm2, mm1
psrld	mm1, 8
por	mm1, mm2
pminub	mm1, mm0

Implementation using MMX Technology

Movq	mm1, mm0
Movq	mm3, mm0
Punpcklbw	mm1, mm0
Punpckhbw	mm3, mm0
Punpckhbd	mm1, mm1
Punpckhbd	mm3, mm3
Punpckhdq	mm1, mm3
Movq	mm2, mm0
Psubusb	mm2, mm1
Pcmpeqb	mm2, mm6
Movq	mm3, mm2
Pand	mm3, mm0
Pandn	mm2, mm1
Por	mm3, mm2

3 Performance

3.1 Gains/Improvements

Streaming SIMD Extensions are an extension to the SIMD concept introduced with the MMX™ technology. The performance improvement results from a combination of MMX technology optimization techniques and utilizing the additional capabilities of the Streaming SIMD Extensions. Increasing parallelism using SIMD and eliminating branches are optimization techniques exploited in the MMX technology. With the introduction of Streaming SIMD Extensions, instructions have been added which can reduce the number of instructions previously required to accomplish the same operation, as well as instructions to improve cache management. The implementation using Streaming SIMD Extensions reduces the number of instructions required compared to the implementation using MMX™ instructions by half, from 14 instructions to 7.

3.1.1 Increase Parallelism Using SIMD and Eliminate Branches

In a C implementation, one RGBA value is processed per loop iteration. Within the loop, there are three sequential “if” statements. For each “if” statement that does not predict the correct outcome, the processor stalls. The instruction pipeline must be cleared and restarted with the correct instructions. A processor stall is a heavy penalty. Unless the data is such that there are very few mis-predictions (*i.e.* the R, G, and/or B values are almost never changed to alpha or they are almost always changed), the C implementation will not perform very well. Using SIMD instructions, two RGBA values (or 8 bytes of data) can be processed per loop iteration; in addition, the difficult-to-predict branch instructions are eliminated. Alternatively, the “if” statement branches could be eliminated in a C implementation using the `cmov`*cc* instruction. However, the `cmov`*cc* instruction is limited to 16 or 32 bit values. The RGBA

values used in this application note are 8 bit quantities. Although the data structure could be altered for 16 bit quantities, the amount of data that could be processed would be cut in half when converted to a SIMD implementation. Comparing a C implementation, just using the SIMD MMX instructions for this algorithm can improve the overall performance between 4x and 8x depending on the cache state. Using Streaming SIMD Extensions can further improve the performance as observed above.

3.1.2 Software Controlled Data Prefetching

A major performance boost is a result of the programmer's ability to influence the processor's cache. Streaming SIMD Extensions provide hints to the processor about what data will be required for future processing. At the same time, the programmer needs to be more aware of when the data is needed, and the potential effect on the cache that may remove some other data to accommodate the new request.

There are two keys to managing the cache:

- Prefetching data from main memory to cache while still doing useful work sufficiently ahead (*i.e.* just in time) of when the processor needs the new data.
- Only prefetching data into cache that will be needed soon without causing the cache to remove other cached data that is also needed in the near term.

In the RGB alpha saturation code, data is prefetched into a non-temporal cache structure 96 bytes ahead of the current data being accessed, and incremented by 16 bytes each iteration through the main loop when unrolled. The reason for prefetching 96 bytes ahead is that the unrolled loop is 24 instructions, which is about 12 clocks without including any memory load latencies in the parallel execution architecture. The main loop processes 4 RGBA values, two RGBA values in each MMX register or 16 bytes, per unrolled loop iteration. The loop takes about 80-120 clocks to process 24 RGBA values, or 96 bytes, in six iterations. Depending on the memory hardware and bus performance, this is approximately the memory load latency. This means that while the data is being loaded into the cache structure, the processor can continue to work on 24 RGBA values before the prefetched data is required by the processor.

3.2 Considerations

3.2.1 Data Alignment

Although not explicitly discussed as part of the algorithm, data alignment is checked at the beginning of each procedure and is assumed to be 32 byte aligned. Thirty-two byte alignment was chosen so that the beginning of the array of RGBA values lines up with a data cache line. A load access to the beginning of the array effectively loads the data into cache for the next four loop iterations in the unrolled implementation using Streaming SIMD Extensions. Although the 32 byte alignment is not absolutely critical to performance, ensuring that data access does not cross cache line boundaries is very critical. Since the data in this application note loads two RGBA values into one MMX register, it is very important that the 8 bytes do not cross a 32 byte cache alignment boundary. If the data does cross the 32 byte alignment boundary, the single load must be converted to two loads for each data cache line split.

3.2.2 Expected Input Data

In this application note, random data was generated for values of R, G, B, and alpha. With this type of data, there is a good chance that one of the R, G, and B values will be greater than alpha. With this in mind, the MMX technology and Streaming SIMD Extensions alpha saturation will always store the

RGBA value back to the array in memory using the `movq` instruction. If it were possible to know that updates to the R, G, and B values were very rare, then it might make sense to use the `maskmovq` instruction. The `maskmovq` instruction does not initiate an actual store unless a given byte needs to be updated. This can reduce the overall memory bandwidth requirement. However, the penalty is that two additional instructions are required.

Frequent Data Updates

```
pminub   mm0, mm1
movq   mmword ptr [eax], mm0
```

Infrequent Data Updates

```
psubusb     mm0, mm1
pcmpeqb     mm0, mm6
pxor        mm0, mm5
maskmovq    mm1, mm0
```

3.2.3 Prefetching and the Cache

Memory accesses, both in the local procedure as well as the full application, should be considered when using the Streaming SIMD Extensions prefetch instructions. If the data is temporary, both in the local procedure as well as the full application, then it makes sense to use the non-temporal prefetch instruction. However, if the data is only temporary from the local procedure viewpoint, but will be used again by another procedure relatively soon, then using the non-temporal load/store instructions is not the best choice. The L2 cache is much larger than the L1 cache and is much faster than main memory. Even if the data is not going to be immediately accessed again in the local procedure, it still may be beneficial to the whole application to keep the data in the L2 cache.

The amount of data accessed is also a consideration when designing an application. If the application processes a large amount of data, like a digital photographic image, it may be more appropriate to design a pipeline to process small portions at a time through different stages. By designing the application in this way and making the portions small enough to remain in cache, the performance can be greatly enhanced. This is in contrast to processing the whole image through each step of the application. A large image is likely to pollute or thrash the data cache since the amount of data to be processed is much larger than the available cache size. As a result, data that is reusable needs to be reloaded from memory in the next process step.

4 Conclusion

Streaming SIMD Extensions can significantly improve performance for integer based algorithms even with applications that have already taken full advantage of the original MMX™ technology integer instructions and optimization techniques. Some of this performance improvement is due to the reduced number of instructions required, and some is due to prefetching. However, the contributions to performance between the reduced number of instructions and prefetching are not independent. The performance improvement is dependent on the contents of the data cache. Instead of a purely perfect or cold cache scenario, it's likely to be a mixture somewhere in between, and so actual performance varies as well. The important point is that the MMX technology optimization techniques still apply. Furthermore, by utilizing the Streaming SIMD Extensions instruction set to reduce the number of instructions, and by effective management of the cache, the performance can be increased even more.

5 C++ Coding Example

```
/* C_alpha_saturation

    Input:  pixels - array of RGBA vectors
             sz     - size of the array
    Output: pixels - modified array of RGBA vectors where any R,G,B value
                     greater than A is replaced with the A from that pixels.

    Alpha saturation is the process of comparing each R, G, B value with an
   associated alpha value.  Any R, G, B value that is greater than alpha
   is replaced with the alpha value.

    For example,  R(255) G (80) B (3) A(79) -> R(79) G(79) B(3) A(79)

*/
#include <stdlib.h>
#include <assert.h>

typedef unsigned char byte;
typedef struct _pixel {
  byte R;
  byte G;
  byte B;
  byte alpha;
} PIXEL;

PIXEL* C_alpha_saturation (PIXEL *pixels, int sz)
{
  // assure optimal memory/cache accesses
  assert( !(((unsigned int)pixels) & 31) );

  for (int i = 0; i < sz; i++)
  {
    PIXEL pix = pixels[i];
    if (pix.R > pix.alpha) pix.R = pix.alpha;
    if (pix.G > pix.alpha) pix.G = pix.alpha;
    if (pix.B > pix.alpha) pix.B = pix.alpha;
    pixels[i] = pix;
  };
  return pixels;
}
```

6 MMX Technology Assembly Code Example

```
/*
  ASM_alpha_saturation
  Input:  pixels - array of RGBA vectors
          sz     - size of the array
   Output: pixels - modified array of RGBA vectors where any R,G,B value
                    greater than A is replaced with the A from that pixels.

    This procedure is an alpha saturation using MMX(tm) instructions.
    Alpha saturation is the process of comparing each R, G, B value with an
associated alpha value.  Any R, G, B value that is greater than alpha is
replaced with the alpha value.

    For example,  R(255) G (80) B (3) A(79) -> R(79) G(79) B(3) A(79)

   The genenral algorithm is to load two vectors of 4 bytes into 1 MMX(tm)
register.  Build a mask of the alpha values in a second MMX(tm) register.
Take the minimum value between the two MMX(tm) registers.  Store the result
back into the pixels array

         R1 G1 B1 A1 R0 G0 B0 A0   where R1 > A1 and G0 > A0
         A1 A1 A1 A1 A0 A0 A0 A0   build mask of alphas
         -----------------------
         A1 G1 B1 A1 R0 A0 B0 A0   result is the minimum of each byte pair

    In the main "for loop", this procedure processes 2 RGBA vectors at a
time.
    Since the size of the array may not be multiple of 2, we need to handle
the case where the size is odd.  If the size of the array is odd (i.e.
remainder is 1), we alpha saturate the last pixels in the array and then the
size is a multiple of 2.
*/

#include <stdlib.h>
#include <assert.h>

typedef unsigned char byte;
typedef struct _pixel {
  byte R;
  byte G;
  byte B;
  byte alpha;
} PIXEL;
```

```
PIXEL* ASM_alpha_saturation (PIXEL *pixels, int sz)
{
  /* Register allocation
     edx - pixels
     ecx - sz

       mm0 - two sets of RGBA pixels values
       mm1 - mask of alpha values
       mm2 - greater than alpha mask
       mm3 - final result (and intermediate values)
       mm6 - 0 (zero)
    */

  // assure optimal memory/cache accesses
  assert( !(((unsigned int)pixels) & 31) );
  __asm
  {
        mov       edx, sz
        mov       esi, pixels
  ; if ((~sz & 1) == 0)
     mov       eax, edx
        xor       eax, -1
        test      eax, 1
        jne       loop
     ; sz = sz -1;
        dec       edx
        xor       eax, eax
        movd      mm0, eax
     ; int* last_vec = (int*) (pixels + sz);
     ; pix  = _m_from_int(*last_vec);
        mov       eax, DWORD PTR [esi+edx*4]
        movd      mm1, eax
        movq      mm2, mm1
     ; tmpalphas1 = _m_punpcklbw(pix, pix);
     ; tmpalphas2 = _m_punpckhbw(pix, pix);
     ; alphas     = _m_punpckhdq (_m_punpckhwd(tmpalphas1, tmpalphas1),
     ;                    _m_punpckhwd(tmpalphas2, tmpalphas2));
        punpcklbw mm1, mm1
        punpckhwd mm1, mm1
        movq      mm3, mm2
        punpckhbw mm2, mm2
        punpckhwd mm2, mm2
        punpckhdq mm1, mm2
```

```
    ; gt_alpha_mask = _m_pcmpeqb(_m_psubusb(pix, alphas), 0);
       movq      mm2, mm3
       psubusb   mm3, mm1
       pcmpeqb   mm3, mm0
    ; *pix = _m_por(_m_pand(gt_alpha_mask, pix),
    ;                         _m_pandn(gt_alpha_mask, alphas));
       pand      mm2, mm3
       pandn     mm3, mm1
       por       mm2, mm3
       movd      eax, mm2
       mov       DWORD PTR [esi+edx*4], eax
loop:
  ; for (int i=0; i < sz; i+=2)
       test      edx, edx
       jle       loopdone
    ; pix  = (__m64*) (pixels + i);
       mov       eax, esi
       lea       ecx, DWORD PTR [esi+edx*4]
forloop:
       xor       edx, edx
       movq      mm0, MMWORD PTR [eax]
       movd      mm1, edx
       movq      mm2, mm0

    ; tmpalphas1 = _m_punpcklbw(pix, pix);
    ; tmpalphas2 = _m_punpckhbw(pix, pix);
    ; alphas     = _m_punpckhdq (_m_punpckhwd(tmpalphas1, tmpalphas1),
    ;                    _m_punpckhwd(tmpalphas2, tmpalphas2));
       punpcklbw mm0, mm0
       movq      mm3, mm2
       punpckhwd mm0, mm0
       punpckhbw mm2, mm2
       punpckhwd mm2, mm2
       punpckhdq mm0, mm2

    ; gt_alpha_mask = _m_pcmpeqb(_m_psubusb(pix, alphas), 0);
       movq      mm2, mm3
       psubusb   mm3, mm0
       pcmpeqb   mm3, mm1

    ; *pix = _m_por(_m_pand(gt_alpha_mask, pix),
    ;                         _m_pandn(gt_alpha_mask, alphas));
       pand      mm2, mm3
```

```
        pandn     mm3, mm0
        por       mm2, mm3
        add       eax, 8
        movq      MMWORD PTR [eax-8], mm2
        cmp       eax, ecx
        jl        forloop

loopdone:
       emms
  }
  return pixels;
}
```

7 Streaming SIMD Extensions Assembly Code Example

```
/*
  Unrolled_ASM_XMM_alpha_saturation
  Input:  pixels - array of RGBA vectors
          sz     - size of the array
  Output: pixels - modified array of RGBA vectors where any R,G,B value
                   greater than A is replaced with the A from that pixels.


This procedure is an alpha saturation using Streaming SIMD Extensions.
Alpha saturation is the process of comparing each R, G, B value with an
associated alpha value.  Any R, G, B value that is greater than alpha is
replaced with the alpha value.


    For example,  R(255) G (80) B (3) A(79) -> R(79) G(79) B(3) A(79)


  The genenral algorithm is to load two vectors of 4 bytes into 1 MMX(tm)
register.  Build a mask of the alpha values in a second MMX(tm) register.
Take the minimum value between the two MMX(tm) registers.  Store the result
back into the pixels array


         R1 G1 B1 A1 R0 G0 B0 A0   where R1 > A1 and G0 > A0
         A1 A1 A1 A1 A0 A0 A0 A0   build mask of alphas
         -----------------------
         A1 G1 B1 A1 R0 A0 B0 A0   result is the minimum of each byte pair


In the main "for loop", this procedure processes 4 RGBA vectors at a time.
This "unrolling" the loop improves the throughput of the parallel pipelines.
Since the size of the array may not be multiple of 4, we need to handle the
case where the size divided by 4 has a remainder of 1,2, or 3.  If the size
of the array is odd (i.e. remainder is 1 or 3), we alpha saturate the last
pixels in the array and then the size is either a multiple of 4 or has a
remainder of 2.  If the size of the array is has a remainder of 2, then we
alpha saturate the last two vectors in the array and then the size must be a
multiple of 4.  We can then process 4 vectors at a time.
```

```
*/
#include <stdlib.h>
#include <assert.h>
#include "mmintrin.h"

typedef unsigned char byte;
typedef struct _pixel {
  byte R;
  byte G;
  byte B;
  byte alpha;
} PIXEL;

PIXEL* Unrolled_ASM_XMM_alpha_saturation (PIXEL *pixels, int sz)
{
  /* Register allocation
     edx - pixels
     ecx - sz

     mm0 - two sets of RGBA pixels values and the final result
     mm1 - mask of alpha values
     mm2 - tmp intermediate value

     mm3 - two sets of RGBA pixels values and the final result
     mm4 - mask of alpha values
     mm6 - tmp intermediate value
     mm7 - mask used in the shuffle process to build mask of alpha values
  */

  // assure optimal memory/cache accesses
  assert( !(((unsigned int)pixels) & 31) );

  __m64 mask = 0xff00ff00ff00ff00;           // mask will be stored in mm7

  __asm
  {
     prefetchnta mmword ptr [pixels]
     mov         edx, pixels
     mov         ecx, sz
     movq        mm7, mask

  ; if ((~sz & 1) == 0)
     mov       eax, ecx
```

```
     xor       eax, -1
     test      eax, 1
     jne       remainder_of_2

     ; sz = sz -1;
     dec       ecx

     ; int* last_vec = (int*) (pixels + sz);
     ; pix  = _m_from_int(*last_vec);
     mov       eax, ecx
     shl       eax, 2
     add       eax, edx
     movd      mm0, DWORD PTR [eax] ; mm0 = pix

     ; alphas = _m_pand (_m_pshufw (pix, 245), 0xff00ff00ff00ff00);
     ; alphas = _m_por (alphas, _m_psrldi(alphas, 8));
     pshufw    mm1, mm0, 0xF5
     pand      mm1, mm7
     movq      mm2, mm1
     psrld     mm1, 8
     por       mm1, mm2

     ; result = _m_pminub (pix, alphas)
     pminub    mm0, mm1
     movd      dword ptr [eax], mm0

   ; if ((~sz & 2) == 0)
remainder_of_2:
     mov       eax, ecx
     xor       eax, -1
     test      eax, 2
     jne       loop

     ; sz = sz - 2;
      add      ecx, -2

     ; pix  = (__m64*) (pixels + sz);
     mov       eax, ecx
     shl       eax, 2
     add       eax, edx
     movq      mm0, mmword ptr [eax]  ; mm0 = pixels(i) and pixels(i+1)

     ; alphas = _m_pand (_m_pshufw (pix, 245), 0xff00ff00ff00ff00);
```

```
    ; alphas = _m_por (alphas, _m_psrldi(alphas, 8));
    pshufw    mm1, mm0, 245
    pand      mm1, mm7
    movq      mm2, mm1
    psrld     mm1, 8
    por       mm1, mm2    ; mm1 = alpha mask

    ; result = _m_pminub (pix, alphas)
    pminub    mm0, mm1
    movq      mmword ptr [eax], mm0

loop:
  ; for (int i=0; i < sz; i+=4)
    test      ecx, ecx
    jle       loopdone

    ; pix  = (__m64*) (pixels + i);
    mov       eax, edx
    lea       ecx, DWORD PTR [edx+ecx*4]

forloop:
    // 0- and 1- indicate the sequence of instructions unrolled.
    // Instructions are mixed so that we don't create bottle necks
    // and improves the pipeline process.

    prefetchnta mmword ptr [eax + 96]  ; sets up the L1 cache
                                       ; Non-temporal is used so we don't
                                       ; destroy the whole cache

    movq      mm0, mmword ptr [eax]       ; 0- mm0 = pix
    pshufw    mm1, mm0, 245               ; 0-
    pand      mm1, mm7                    ; 0-

    movq      mm3, mmword ptr [eax + 8]   ; 1- mm3 = pix
    pshufw    mm4, mm3, 245               ; 1-

    movq      mm2, mm1                    ; 0-
    psrld     mm1, 8                      ; 0-
    por       mm1, mm2                    ; 0- mm1 = alphas
    pminub    mm0, mm1                    ; 0- mm0 = final result
    movq      mmword ptr [eax], mm0       ; 0-

    pand      mm4, mm7                    ; 1-
```

```
     movq      mm5, mm4                   ; 1-
     psrld     mm4, 8                     ; 1-
     por       mm4, mm5                   ; 1- mm4 = alphas

     pminub    mm3, mm4                   ; 1- mm3 = final result
     movq      mmword ptr [eax + 8], mm3  ; 1-

     add       eax, 16
     cmp       eax, ecx
     jl        forloop
loopdone:
     emms
  }
  return pixels;
}
```